ANL/NE-17/31

ARC integration into the NEAMS Workbench

Status and verification of the ARC neutronic fast reactor tools integration to the NEAMS Workbench

Nuclear Engineering Division

ANL/NE-17/31

ARC integration into the NEAMS Workbench

Status and verification of the ARC neutronic fast reactor tools integration to the NEAMS Workbench

prepared by
N. Stauff, N. Gaughan, and T. Kim
Nuclear Engineering Division
Argonne National Laboratory

September 30, 2017

ABSTRACT

One of the objectives of the Nuclear Energy Advanced Modeling and Simulation (NEAMS) Integration Product Line (IPL) is to facilitate the deployment of the high-fidelity codes developed within the program. The Workbench initiative was launched in FY-2017 by the IPL to facilitate the transition from conventional tools to high fidelity tools. The Workbench provides a common user interface for model creation, real-time validation, execution, output processing, and visualization for integrated codes.

This report details the efforts under way for integrating the Argonne Reactor Computation (ARC) suite of codes into the Workbench. The ARC codes contain both legacy codes like DIF3D and REBUS-3 that were developed with over 30 years of experience, and newer NEAMS additions like MC^2-3 and PERSENT. These codes are extremely attractive by their flexible capabilities and computational efficiency. However, they require knowledge of reactor physics and experience on fast reactor design in order to be familiar with the extent of their capabilities. The ARC codes employ an inconvenient input system, and users mostly rely on scripts, developed based on their experiences, to generate inputs. For these reasons, it was decided to integrate the ARC codes within the NEAMS Workbench, and to provide the user with a new common input allowing to build a core model and to describe the calculations requested.

This new type of integration into the Workbench was successfully demonstrated through this project as the MC^2-3, DIF3D, REBUS-3, and PERSENT codes can be used through the Workbench for solving real problems. A fast reactor type of geometry can be modeled through the Workbench, as demonstrated with a simple benchmark problem. However, some advanced calculation methodologies such as heterogeneous cross-section treatment in MC^2-3 and equilibrium burnup calculation in REBUS-3 could not be implemented at this time and should be the focus of future effort.

Integrating the ARC codes into the Workbench benefits directly the ARC community by providing a set of controlled, maintained and validated scripts to generate ARC inputs, which promotes best practices, and facilitates learning how to use the codes. The second benefit from this project results directly from taking advantage of the capabilities of the Workbench interface to improve the user experience with the ARC codes. The ARC codes are currently used through the Workbench by nuclear engineers at ANL and new users from universities will be trained in early FY-2018 as future efforts will focus on building some user experience.

This project directly benefits the NEAMS program as it favors using the NEAMS codes such as MC^2-3 and PERSENT. Future work should also focus on integrating high-fidelity codes such as PROTEUS into the Workbench, which should be performed using the same common input logic developed for the ARC codes.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

1. Introduction

One of the objectives of the Nuclear Energy Advanced Modeling and Simulation (NEAMS) Integration Product Line (IPL) is to facilitate the deployment of the high-fidelity codes developed within the program. The Workbench [1] initiative was launched in FY-2017 by the IPL to facilitate the transition from conventional tools to high fidelity tools. The Workbench provides a common user interface for model creation, real-time validation, execution, output processing, and visualization for integrated codes. This report summarizes the significant efforts put toward implementing the Argonne Reactor Computation (ARC) suite of codes into the NEAMS Workbench.

The ARC suite of codes has been developed at ANL since the 1980's for fast reactor analysis. It gathers neutronics, thermal hydraulics, safety, and fuel behavior analysis codes, as displayed in Figure 1-1. The current focus of this study is on the ARC suite of deterministic neutronic codes, which contains the MC^2-3 code for multi-group cross-section processing, the DIF3D code for flux calculation, REBUS-3 for depletion and equilibrium calculations, and PERSENT for perturbation theory calculations.

These ARC codes are used at national labs, universities, and companies for fast reactor analysis. They gather more than 30 years of development, went through extensive validation and verification, and can solve complex physics phenomena in a very efficient way. However, these codes require knowledge on reactor physics and experience on fast reactor design in order to be familiar with the extent of their capabilities. The ARC codes employ an inconvenient input system, and users mostly rely on scripts, developed based on their experiences, to generate inputs.

Integrating the ARC codes into the Workbench was initiated to address these challenges and to improve user experience with these codes by taking advantage of the various benefits brought by the Workbench interface. This report is intended for new users of the ARC codes within the Workbench, and for code developers that will integrate their codes into the Workbench. In Section 2, the method used for the implementation is detailed together with the list of developments that were needed. Section 3 describes the status of the implementations of the different ARC codes. A tutorial for users of the ARC codes through the Workbench was developed in Appendix A. Results from a simple benchmark problem are summarized in Section 4 for demonstration purposes. Finally, Section 5 provides the main conclusions.

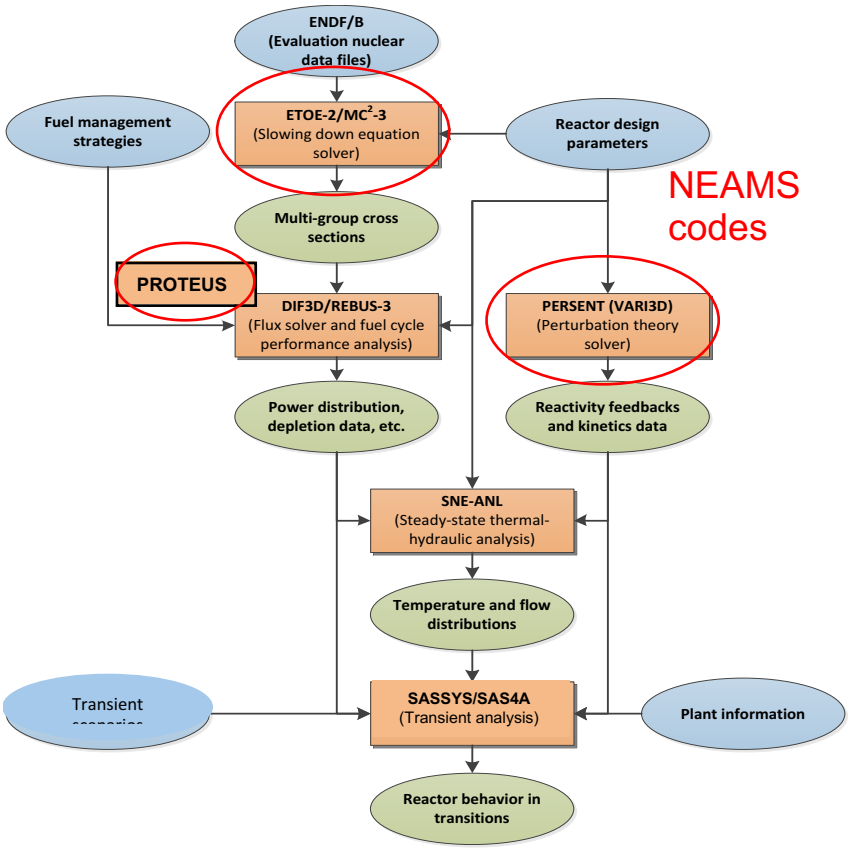

Figure 1-1. The Argonne Reactor Computation suite of codes.

2. Framework for ARC integration

The first question arising when integrating a code into the Workbench is whether one wants the user to specify the code's input data using the code's native input format, or provide a different input format. Giving the user direct access to the native input of the code requires a "white box" type of integration where the Workbench can use the code's own input processing logic modules, while developing a new input format can require a "black box" type of integration where the Workbench must rely on an opaque code module that conducts the native input formatting. Very early on in this project, it was decided to use a "black box" type of implementation for the ARC codes. This is generally the natural solution when dealing with legacy codes with inconvenient inputs. In the case of the ARC codes, it is also consistent with the way these codes are used by nuclear engineers relying on scripts to help them process their different codes' inputs.

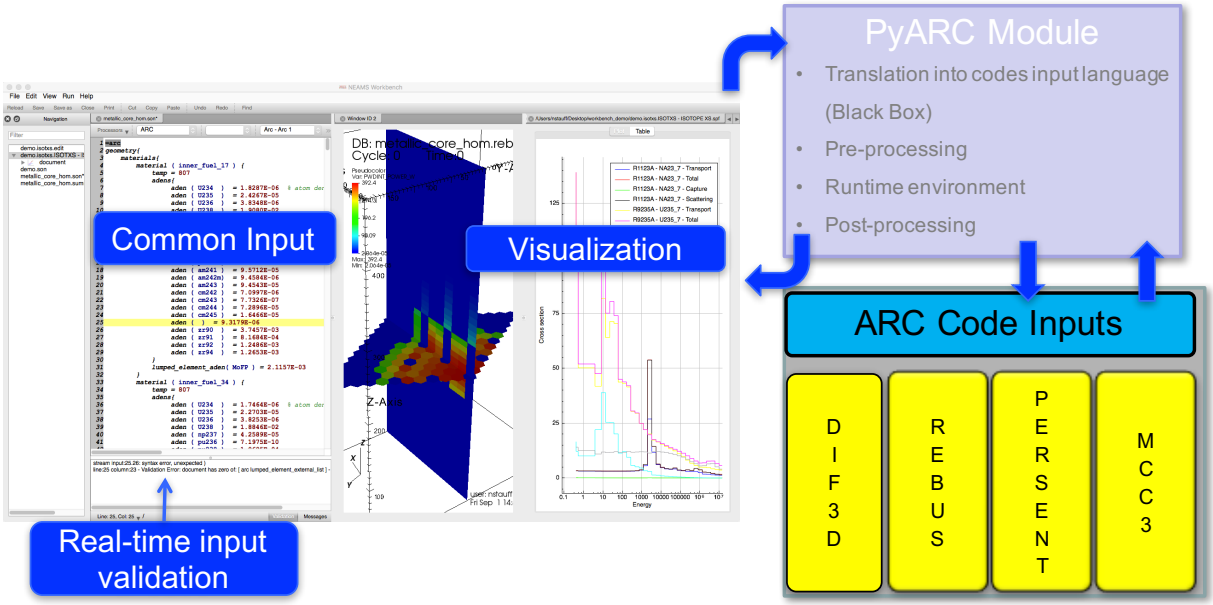

Figure 2-1. Structure of the ARC integration in the Workbench.

The structure of the Workbench integration is illustrated in Figure 2-1: there are several components to the integration that are described in this section:

- Workbench interface: It is developed at ORNL and designed to assist new users, while not obstructing experienced users. Several components of this interface are required for a code's integration:
  - Common input
  - Templates
  - Visualization

- PyARC module: this is a python module required for "black box" integration that contains the logic for processing the code's inputs, running them, and post-processing the outputs. This module is the glue between the Workbench interface and the ARC codes.

2.1 Workbench Interface

2.1.1 Common input

Any type of code integration into the Workbench requires developing a "schema". This document describes the structure and logic, or the definition, of the input, enabling the real-time validation and auto-completion capabilities of the Workbench. In the case of a "black box" integration, the schema describes the new type of input that will be seen by the user. This input is described as "common input" as it is used to generate inputs for MC^2-3, DIF3D, REBUS-3, and PERSENT for integrated problem-dependent cross-section preparation, core analysis, depletion, and sensitivity/uncertainty analysis. The "arc.sch" schema was developed in close collaboration between ANL and ORNL teams and contains **~2000 lines of input definition**. Both the common input and input schema are formatted in Standard Object Notation (SON) format, described in Figure 2-2.

```
% SON unit of execution block
%     terminated by 'end' in first column
=program_name

  % SON Keyed-value
  name = value

  % SON Object - delimited by '{' '}'
  object_name {
     % Object contents - keyed values, objects, and arrays
     x = 1 y = 2
     z = 3
  }

  % SON Array - delimited by '[' ']'
  array_name [
    % array contents - scalar values, keyed values, objects, or arrays
    1 2 3
  ]
end
```

Figure 2-2. SON format description.

This "common input" allows modeling a reactor geometry in an intuitive and flexible way. It was developed with continuous involvement of ARC users at ANL and is continuously going through some improvements after receiving first user's feedbacks. For now, only fast-reactor types of

geometries (with a hexagonal lattice) is allowed for ARC codes use. However, should future integrated codes adopt this common input structure, additional types of geometries can be easily enabled or implemented. The structure of the input is summarized in Figure 2-3 and a tutorial was developed in Appendix A to explain in detail the input logic.

```
=arc
geometry{
% Description of the core model
    materials{
% List of materials (in atom/weight fractions, densities or formulas)
    }
    blends{
% List of mixtures of materials for homogenized modeling
    }
    surfaces{
% List of surfaces used for geometry description (hexagon, plane, cylinder)
    }
    regions_reactor{
% Description of the core, with lattice of assemblies and description of
% each type of assembly. Assemblies can be defined stacking several axial
% sub-assemblies described as homogeneous regions or with explicit geometry
    }
}
calculations{
% Contains the list of calculations
    mcc3{
% Description for MCC3 calculations
    }
    dif3d{
% Description for DIF3D calculations
        rebus{
% Description for REBUS calculation (based on DIF3D options)
        }
        persent{
% Description for PERSENT calculations (based on DIF3D options)
        }
    }
}
end
```

Figure 2-3. Structure of the common input.

2.1.2 Templates

The Workbench contains the HierarchicAL Input Template Engine (HALITE) developed to expand hierarchical input data into code-specific input. Due to the complexity of the ARC native

inputs, it was decided for now to push the writing logic of the ARC native input logic into the PyARC module. Templates are still used to assist users in generating the common input within the Workbench. The common input templates were developed in parallel to the schema and the common input. Those are blocks of input with default values accessible for convenience to the user. A total of **73 templates** were generated for the ARC codes. For instance, the user can directly access input blocks for different decay chains to fill out the REBUS-3 input, through the available templates.

2.1.3 Visualization

The Workbench provides two built-in types of visualization capabilities that the ARC integration could take advantage of, providing little development:

- VisIT [11]

Post-processing capabilities were added in the past to DIF3D and integrated in PERSENT for generating "vtk" files for results visualization with the VisIT software. VisIT being integrated into the Workbench allows direct visualization of the ARC post-processed outputs, as illustrated in Figure 2-1. Some development was needed in the runtime environment to allow generating the "vtk" files for DIF3D, REBUS-3 and PERSENT calculations.

- Built-in plotting capability

The Workbench interface supports plotting capabilities using line plots, histograms, bar charts, etc. Two types of line plots were implemented in collaboration with ORNL to display the multi-group cross sections processed by MC^2-3 (as also illustrated in Figure 2-1), and the region-wise flux spectrum printed by DIF3D and REBUS-3. Such plotting implementation requires developing a new "processor" file which often uses familiar command line utilities such as Grep and Awk to extract data to plot from the codes' output. The processor files provide code integrators or regular users a means to extend Workbench post-processing visualization capabilities.

***2.2** PyARC*

As illustrated in Figure 2-1, the PyARC module is the glue between the Workbench interface and the ARC codes. For a "black box" integration, this glue is essential as it contains the logic to:

- extract information from the common input entered through the Workbench
- perform additional verifications that the validation engine of the Workbench cannot perform (for instance checking that the link to an input containing cross-sections or a decay chain is valid)
- pre-process the information, calculating for instance homogenized atom densities in different regions
- generate the ARC codes' inputs
- handle the runtime environment, for instance running MC^2-3 elementary cell calculations in parallel on different CPUs

- post-process the outputs, printing out summary files with main results of the different codes' outputs.

The PyARC module is developed in Python through a collaborative environment on GitLab at code.ornl.gov so that new additions are tracked and reviewed. Pylint is used during the development process to check compatibility with PEP8 coding standards. This module is developed under the leadership of ANL with the assistance and contribution of the Workbench team at ORNL. The whole module contains ~**3000 lines** of python coding. The PyARC module relies on the following sub-modules:

- PyARCModel: loads the input, performs list of additional verification, performs pre-processing on the input.
- PyARCUtils: contains utilities procedures
- PyARCUserObject: defines variables and procedures that are used throughout the code
- PyMCC3, PyTwoDant, PyREBUS, PyPERSENT: contain the logic for input writing, execution, and post-processing for each code
- The PyARC module also relies on the PySCL module that is developed by ORNL to provide the standard composition library (SCL) in a consistent way between different codes

Unit tests are developed for regression testing after each code modification and prior to committing and pushing modifications on the main branch. Currently, **45 unit tests** are implemented to check the common input processing, interpretation of the standard composition library, input generation of MC^2-3, DIF3D, REBUS-3, TwoDant, PERSENT, and post-processing of the outputs. Consequently, the unit tests check the pre-processing, input writing, and post-processing logic of PyARC. The execution logic is currently tested through manual runs but will be integrated into a continuous integration test suite in the future that also includes integration and system tests of ARC codes execution.

3. Status of the integration

One of the benefit of the "black box" type of integration is that the user is shielded from the original input of the legacy codes. The associated challenge is that some of the options and code's capabilities may not be made available to the user through the "black box". The ARC integration using such a "black box" approach focuses on the most popular and important capabilities of each code. This chapter reviews the status of the ARC integration, listing the capabilities available for each code, and which capabilities will be integrated in the future, as also summarized in Table 3-1.

For any calculation, the user is provided with the ARC input files generated, the full output files, and a summary file that contains some post-processed data extracted for user-convenience purposes. Other files such as the ISOTXS and "vtk" files are returned as well.

Table 3-1. Summary of the ARC integration.

Codes	Capabilities implemented in FY17	Future developments suggested
MC^2-3	-Homogeneous calculation -Region-wise flux condensation using TwoDant	-Heterogeneous calculations -Gamma XS and delayed neutron constants
DIF3D	-Hex-z model -VARIANT, Nodal, FD	-Other geometries -Gamma calculations
REBUS-3	-Once-through depletion calculations	-Equilibrium calculations
PERSENT	-Perturbation and Sensitivity calculations on eigenvalues, with option to update cross-sections	-Perturbation and Sensitivity calculations on additional problems with depleted compositions -Uncertainty calculations

3.1 MC^2-3

The MC^2-3 code is developed within the NEAMS program for multi-group cross-section processing into an ISOTXS binary file. From the Workbench, one currently can generate cross-sections for pre-generated or user-defined energy-group structure, on a mixture-type geometry, with different scattering orders. The user can lump cross-sections together to define lumped fission products for instance, or use pre-defined lumped elements.

For region-wise flux condensation, two approaches were implemented with the Workbench. The first approach consists of generating neutron leakage files from the critical regions that can be used as external sources in the sub-critical regions. The second approach consists of using TwoDant, which is a SN neutron solver, for fine-mesh flux calculation using an equivalent 2D (RZ) core model.

Future work should focus on implementing the heterogeneous-type of modeling capabilities available with MC^2-3 (based on 1D cylindrical or plate geometries) to provide enhanced accuracy to the multi-group cross-section generation. Calculation of delayed neutron constants and gamma cross-sections will also be required in the future.

In terms of output processing, the multi-group cross-sections generated in the ISOTXS file can be plotted automatically in the Workbench interface using a specific "ISOTXS – ISOTOPE XS" processor developed at ORNL, as illustrated in Figure 3-1.

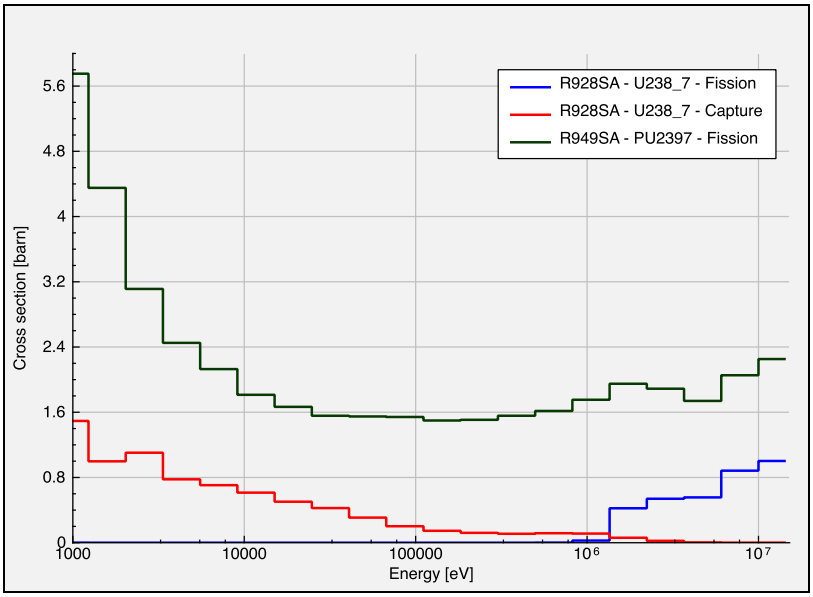

Figure 3-1. Multi-group XS from ISOTXS file calculated and plotted by the Workbench.

3.2 DIF3D

The DIF3D code is a legacy code used for neutron and gamma flux calculations on various types of geometries, based on pre-generated cross-sections. Currently, only neutron flux calculations are integrated into the Workbench. The multi-group cross-sections can be generated using MC^2-3 calculations or a compatible set of previously calculated multi-group cross-sections. While DIF3D allows modeling various types of geometries (hexagonal, cylindrical, Cartesian…), only the hexagonal-z types of geometries are currently supported through the Workbench since those are the most popular ones used with DIF3D to model advanced reactors. The DIF3D code includes 3 neutron solvers (Nodal, FD (Finite Difference), and Variant) that were all enabled with the Workbench.

Post-processing of DIF3D input was initiated by printing out the main information available for every area (defined as the whole core and every sub-assembly region) in the main summary file. The multi-group neutron flux spectrum can be plotted automatically, as shown in Figure 3-2, using the "Flux Spectrum" processor developed at ORNL. The *dif3d_to_vtk* utility code is automatically run after DIF3D execution to generate the "vtk" files allowing direct visualization

of the power density, neutron flux, atom densities, etc., using VisIT through the Workbench, as also shown in Figure 3-2.

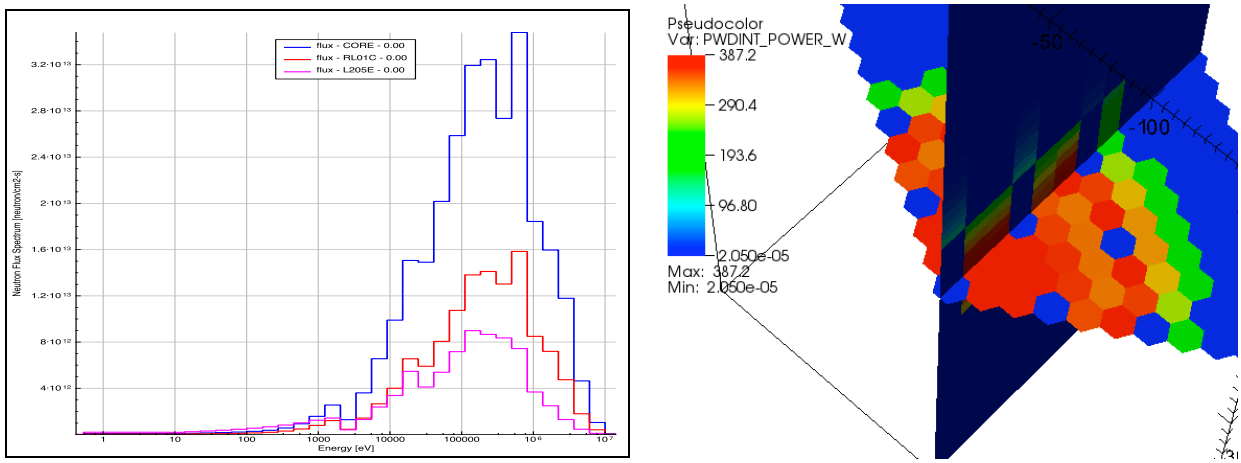

Figure 3-2. Example of visualizations available for DIF3D and REBUS-3: neutron flux spectrum (left) and power map (right) calculated and plotted with the Workbench.

3.3 REBUS-3

REBUS-3 is a legacy code used for depletion calculations using DIF3D solvers that allows a wide range of modeling options such as fuel shuffling, equilibrium and enrichment search. For preliminary implementation of REBUS-3 into the Workbench, only the once-through depletion capability was integrated. The main challenge associated with integrating REBUS-3 was to allow a user-specified decay chain, without making the Workbench input too complicated. This was achieved by providing an external text file containing the decay chain input from REBUS-3 (cards 09, 24, 25), which is being parsed in PyARC. Future work should focus on including the enrichment and equilibrium search capabilities. In terms of post-processing, the same capabilities developed for DIF3D are made available with REBUS-3 at every time-step of the depletion calculation.

3.4 PERSENT

PERSENT is a perturbation theory code developed within the NEAMS program and based on the neutron transport equation in a Hex-Z geometry. It allows calculating feedback coefficients, sensitivity coefficients, and nuclear data uncertainties (when using a covariance matrix). Its integration was initiated in FY-17 by preparing the common input and associated templates, and implementing perturbation and sensitivity calculations on eigenvalue problems. The user can define which materials are perturbed with a change in density or in temperature. The cross-section of the perturbed composition can be automatically re-calculated both for perturbation and for sensitivity calculations.

For illustration purposes, Figure 3-3 shows the distribution of the sodium void worth calculated on an SFR design and plotted within the Workbench. Table 3-2 displays the sensitivity of the

cross-sections of the main isotopes on the eigenvalue in an SFR core, as calculated and summarized with the Workbench.

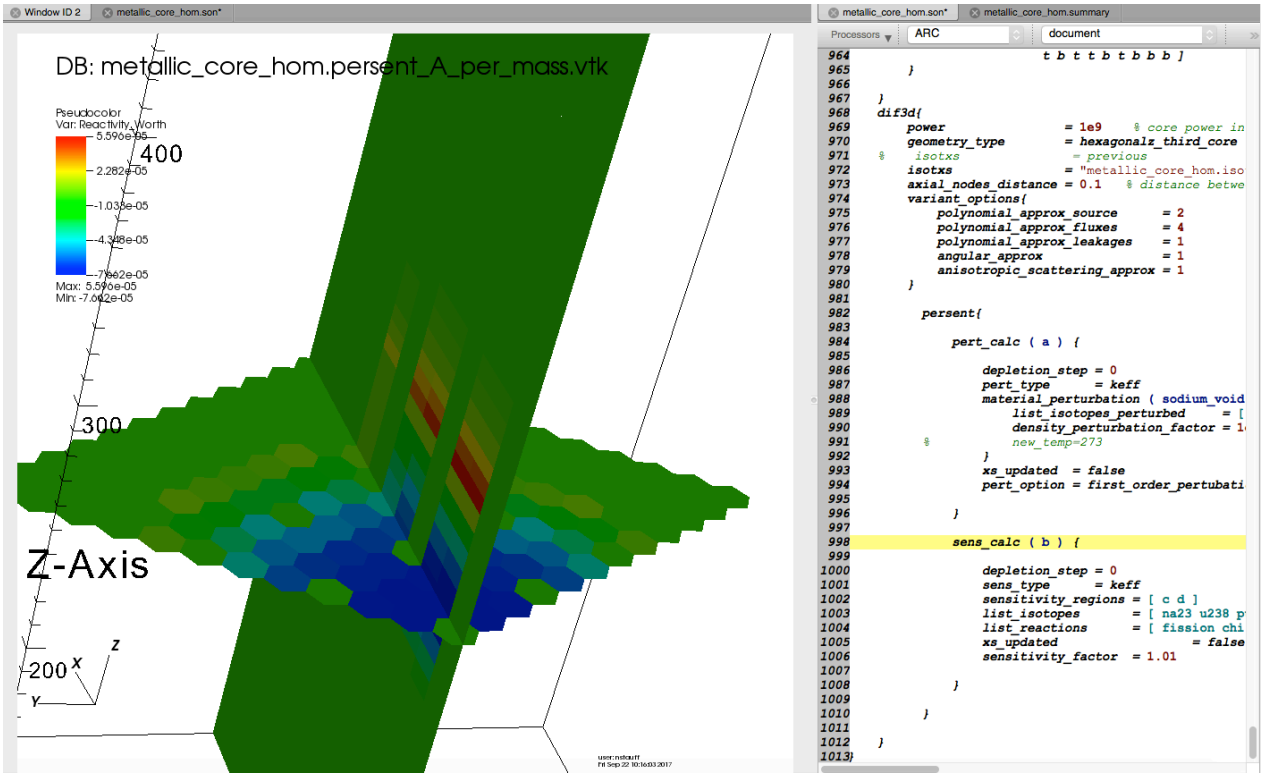

Figure 3-3. Sodium void worth distribution [kg^{-1}] plotted and calculated by the Workbench.

Table 3-2. Main isotopic and reaction breakdown of the sensitivity coefficients on the k-effective of an SFR core, calculated and summarized by the Workbench.

	ν	Fission	Capture	Elastic	Inelastic	χ	TOTAL
Na23				-0.01%	-0.01%		-0.02%
Fe56			-0.01%	0.01%	-0.03%		-0.04%
U238	0.11%	0.07%	-0.15%	0.02%	-0.06%	0.11%	0.10%
Pu239	0.64%	0.47%	-0.03%			0.64%	1.71%

In the future, perturbation theory calculations using PERSENT should be allowed on additional problems (delayed neutron fraction…) and at different time-steps (provided by REBUS-3). Uncertainty quantification will also be integrated. The framework for this was enabled in the PyARC module so that the workflow and input creation will be implemented in the future.

4. Verification with a benchmark problem

Verification of the proper implementation of the ARC codes into the NEAMS Workbench is being conducted with unit tests as discussed in Section 2.2. Those check on different examples that the ARC inputs are properly generated and the correct information is extracted from the outputs. The objective of this section is rather to demonstrate the usefulness of the ARC Workbench implementation by applying it to solve an international benchmark problem.

The OECD/NEA sub-group on Uncertainty Analysis in Modelling (UAM) for Design, Operation and Safety Analysis of Sodium-cooled Fast Reactors (SFR-UAM) has been formed under the NSC/WPRS/EGUAM and specified a series of benchmarks [12, 13]. For demonstration purposes of the already implemented capabilities available in the Workbench, the fuel assembly depletion benchmark proposed within the SFR UAM was modeled using the ARC codes through the Workbench. The description of the oxide-fueled assembly model is detailed in [12]. Preliminary results from 9 contributing participants for this depletion benchmark were published in [13] and the eigenvalue depletion evolution is shown in Figure 4-1. Other parameters, such as the neutron flux, reactivity coefficients, and isotopic compositions are compiled as well, but were not considered in this work.

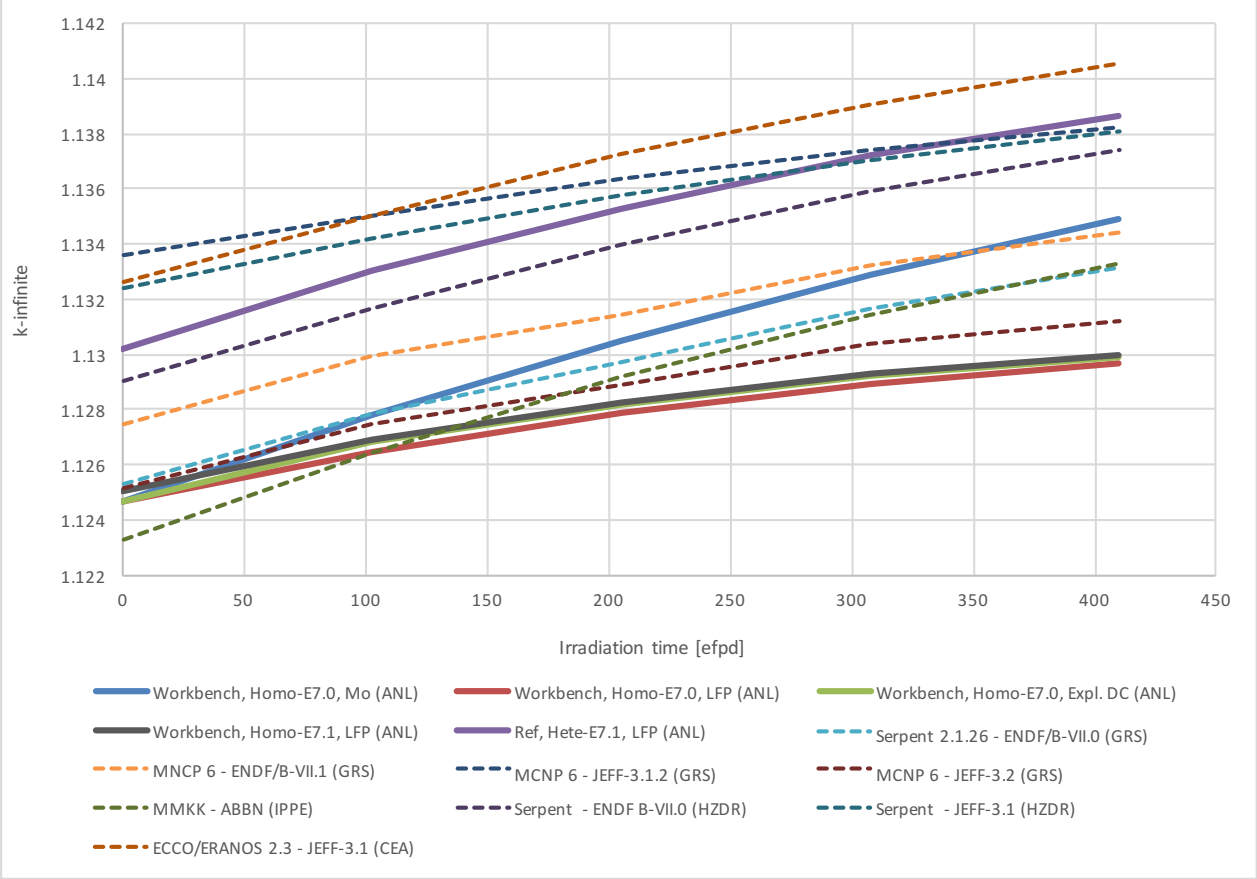

Figure 4-1. Eigenvalue burnup evolution comparison within participants of the SFR UAM assembly depletion benchmark.

ANL contributed to this benchmark using the ARC codes with reference calculation route (described as "Ref, Hete-E7.1, LFP" in Figure 4-1), with heterogeneous processing of the cross-sections from ENDF/B-VII.1, and its reference decay chain based on lumped fission products (LFP). Updated results obtained with the Workbench using various nuclear data libraries (ENDF/B-VII.0 and .1) and decay chains (based on Mo for representing fission products, on reference LFP, and on an explicitly detailed decay chain) were added to this comparison in Figure 4-1 and show consistent results with other participants. However, these updated results display a lower k-infinite of 435 pcm when compared to reference ANL results. This discrepancy is primarily explained by the lack of heterogeneous cross-section treatment currently available with MC^2-3 in the Workbench.

Future benchmark analysis should include the full core SFR analysis described in the SFR UAM benchmark specification [12], and should be conducted once the 1D heterogeneous cross-section treatment of MC^2-3 is implemented in the Workbench to display improved agreement with other participants.

5. Conclusions and Future work

The Workbench initiative was launched in FY-2017 within the NEAMS program to facilitate the transition from conventional tools to high-fidelity tools, employing a bottom-up approach. The Workbench provides a common user interface for model creation, real-time validation, execution, output processing, and visualization for integrated codes.

This report details the efforts under way for integrating the Argonne Reactor Computation (ARC) suite of codes into the Workbench. The ARC codes contain both legacy codes like DIF3D and REBUS-3 that were developed with over 30 years of experience, and newer NEAMS additions like MC^2-3 and PERSENT. These codes are extremely attractive by their flexible capabilities and computational efficiency. However, they require knowledge on reactor physics and experience on fast reactor design in order to be familiar with the extent of their capabilities. The ARC codes employ an inconvenient input system, and users mostly rely on scripts, developed based on their experience, to generate inputs. For these reasons, it was decided to integrate the ARC codes within the NEAMS Workbench, and to provide the user with a new common input allowing to build a core model and to describe the calculations requested.

This new type of integration into the Workbench was successfully demonstrated through this project as the MC^2-3, DIF3D, REBUS-3, and PERSENT codes can be used through the Workbench for solving real problems. For demonstration purposes, these codes were used through the Workbench for modeling a Fast Reactor type of geometry and for solving the SFR-UAM assembly depletion benchmark. However, some advanced calculation methodologies such as heterogeneous cross-section treatment in MC^2-3 and equilibrium burnup calculation in REBUS-3 could not be implemented at this time and should be the focus of future effort. Additional work is also needed to continue the PERSENT integration that was initiated this year.

Integrating the ARC codes into the Workbench benefits directly the ARC community by providing a set of controlled, maintained and validated scripts to generate ARC inputs, which promotes best practices, and facilitates learning how to use the codes. The second benefit from this project results directly from taking advantage of the capabilities of the Workbench interface to improve the user experience with the ARC codes: the Workbench provides assistance for building an input through auto-completion, real-time validation, and access to templates, and for post-processing the output with access to visualization. The ARC codes are currently used at ANL through the Workbench by nuclear engineers for SFR core design analysis and by a summer intern to conduct some benchmark exercises. University faculty from Penn State University and North Carolina State University expressed interest in participating in the early user group and will be trained in early FY-2018. Future efforts should focus on building a broader user experience.

This project directly benefits the NEAMS program as it favors using the NEAMS codes such as MC^2-3 and PERSENT. Future work should also focus on integrating high-fidelity codes such as PROTEUS into the Workbench, which should be performed using the same common input logic developed for the ARC codes.

References

[1] B. T. Rearden, R. A. Lefebvre, A. B. Thompson, B. R. Langley, N.E. Stauff, "Introduction to the Nuclear Energy Advanced Modeling and Simulation Workbench," M&C 2017, Jiju Island, South Korea, April (2017).

[2] C. H. Lee and W. S. Yang, "Development of Multigroup Cross Section Generation Code MC2-3 for Fast Reactor Analysis," FR'09, Kyoto, Japan, Dec. 7-11 (2009).

[3] A. C. Kahler et al., "ENDF/B-VII.1 Neutron Cross Section Data Testing with Critical Assembly Benchmarks and Reactor Experiments," Nuclear Data Sheets, 112, pp. 2997-3036 (2011).

[4] C. H. Lee, N. E. Stauff, "Improved Reactivity Worth Estimation of MC2-3/DIF3D in Fast Reactor Analysis," Proceedings of ANS Sumer Meeting, paper 14201, San Antonio, Texas (2015).

[5] K. L. Derstine, "DIF3D: A Code to Solve One-, Two-, and Three-Dimensional Finite Difference Diffusion Theory Problems," ANL-82-64, Argonne National Laboratory (1984).

[6] G. Palmiotti et al, "Variational nodal transport methods with anisotropic scattering," Nuclear Science and Engineering, Vol. 115, pp. 233-243 (1993).

[7] M. A. Smith, W. S. Yang, A. Mohamed, E. E. Lewis, "Perturbation and Sensitivity Tool Based on the VARIANT Option of DIF3D," ANS Transactions 107, San Diego, Nov. 11-15 (2012).

[8] G. Aliberti and M. Smith, "PERSENT: need of a deterministic code for sensitivity analysis in 3D geometry and transport theory," Proceedings of PHYSOR2014, Kyoto, Japan (2014).

[9] W. S. Yang, T. J. Downar, "Depletion Perturbation Theory for the Constrained Equilibrium Cycle," Nuclear Science and Engineering, 102, pp. 365-380 (1989).

[10] B. J. Toppel, "A User's Guide to the REBUS-3 Fuel Cycle Analysis Capability," ANL-83-2, Argonne National Laboratory (1983).

[11] LLNL: VisIT Visualization Tool (2002– 2016). https://wci.llnl.gov/codes/visit

[12] Benchmark for Neutronic Analysis of Sodium-cooled Fast Reactor Cores with Various Fuel Types and Core Sizes, OECD Nuclear Energy Agency, February 2016, NEA/NSC/R(2015)9.

[13] Gerald Rimpault et al, "Objectives and Status of the OECD/NEA sub-group on Uncertainty Analysis in Best-Estimate Modelling (UAM) for Design, Operation and Safety Analysis of SFRs (SFR-UAM)," FR'17, Yekaterinburg, Russia.

Appendix A: Tutorial

Workbench files are made up of a series of blocks containing **keywords** (bolded in this tutorial). Most keywords require other keywords to follow them, creating 'levels' of keywords within keywords. When a keyword with a higher level is selected by autocomplete, a pair of brackets will appear along with the keyword. All other keywords that come below must be inside these brackets. The user can access the list of keywords available in a block through the auto-completion. It is recommended that the user keeps the end brackets lined up with the corresponding keyword and indent for each new (lower) level. The main structure of the common input is presented in Figure 0-1 and each block/sub-block is presented in this tutorial.

```
=arc
geometry{
% Description of the core model
    materials{
% List of materials (in atom/weight fractions, densities or formulas)
    }
    blends{
% List of mixtures of materials for homogenized modeling
    }
    surfaces{
% List of surfaces used for geometry description (hexagon, plane, cylinder)
    }
    regions_reactor{
% Description of the core, with lattice of assemblies and description of
% each type of assembly. Assemblies can be defined stacking several axial
% sub-assemblies described as homogeneous regions or with explicit geometry
    }
}
calculations{
% Contains the list of calculations
    mcc3{
% Description for MCC3 calculations
    }
    dif3d{
% Description for DIF3D calculations
        rebus{
% Description for REBUS calculation (based on DIF3D options)
        }
        persent{
% Description for PERSENT calculations (based on DIF3D options)
        }
    }
}
end
```

Figure 0-1. Structure of the common input.

1. geometry

The **geometry** block is where the user defines the geometry modeled. It contains three mandatory sub-levels: **materials**, **surfaces** and **regions_reactor**. The **blends** block is not mandatory but can also be defined within **geometry**. This section describes each of these sub-blocks.

1.1 materials

The **materials** block is mandatory as at least one **material** sub-block should be defined. A **material** sub-blocs requires a name (which should be unique) and a temperature (in Kelvin). Workbench allows several different ways to define the isotopic content of a **material**: with weight fractions, atom fractions, atom densities, weight densities, and atom formula. Different types can be used within the same input.

- **wfrac** : weight fractions
- **afrac** : atom fractions
- **adens** : atom densities
    - **lumped_element_aden** : atom density of lumped fission products defined in calculations/mcc3/lumped_element_text_file
- **wdens** : weight densities
- **aform** : atom formula. **aform** is useful for compounds.

```
material ( material_name ) {
    adensity = 0            % total atom density in at/barn-cm
    temp     = 273          % temperature in Kelvin
    aform( 3 ){             % atom count
        afrac ( id1 ) = 0.0  % atom fraction sum to one
        afrac ( id2 ) = 0.0  % atom fraction sum to one
        afrac ( id3 ) = 0.0  % atom fraction sum to one
    }
    aform( 4 ){             % atom count
        afrac ( id1 ) = 0.0  % atom fraction sum to one
        afrac ( id2 ) = 0.0  % atom fraction sum to one
        afrac ( id3 ) = 0.0  % atom fraction sum to one
    }
}
```

In “aform (x)” x=the number of that atom in the compound

Example 1: UO2 with 7% enriched uranium

```
material ( UO2 ) {
    adensity = 1.0e-2           % total atom density in at/barn-cm
    temp     = 273          % temperature in Kelvin
    aform( 1 ){             % atom count
        afrac ( U235 ) = 0.07  % atom fraction sum to one
        afrac ( U238 ) = 0.93  % atom fraction sum to one
    }
    aform( 2 ){             % atom count
        afrac ( O16 ) = 0.0  % atom fraction sum to one
    }
}
```

1.2 blends

The **blends** block is not mandatory but can be used to combine materials, which can be useful for homogenized compositions.

*Example 2: Blends could be used as the material in **assembly_hex** to define homogenized compositions. They require definition of individual materials (fuel_material, sodium, cladding, structure, and b4c).*

```
blends{
    blend ( fuel_blend ) {
         volf ( fuel_material ) = 0.45  % volume fraction sum to less than or equal to one
         volf ( sodium ) = 0.28  % volume fraction sum to less than or equal to one
         volf ( cladding ) = 0.12  % volume fraction sum to less than or equal to one
         volf ( structure ) = 0.08  % volume fraction sum to less than or equal to one
    }
    blend ( control_blend ) {
         volf ( sodium ) = 0.43  % volume fraction sum to less than or equal to one
         volf ( B4C ) = 0.25  % volume fraction sum to less than or equal to one
         volf ( structure ) = 0.24  % volume fraction sum to less than or equal to one
    }
}
```

1.3 surfaces

The first step toward building the geometry is to define surfaces. The choices currently available are hexagons with **axis**=y (the y axis intersects two opposite vertices of the hexagon and the hexagonal prism extends vertically in the z direction, see Figure 1-1), cylinders along the z axis, and planes perpendicular to the z axis. For hexagons, "pitch" means flat-to-flat distance. All dimensions are in meters. Boundary conditions can be specified for each surface: extrapolated (default), reflective, periodic, and vacuum. These conditions may be over-written by the option used in calculations/dif3d/geometry_type.

1.4 regions_reactor

After **surfaces**, the next keyword is **regions_reactor**, which is required. Here the **lower_axial_surf** and **upper_axial_surf** refer to the entire system (other axial sections can be defined later under **sub_assembly_hex**).

```
regions_reactor{
    lower_axial_surf  = plane_name
    upper_axial_surf  = plane_name

    % insert core_hexlattice

    assembly_hex ( assembly_hex_name ) {

    }
}
```

1.4.1 core_hexlattice

core_hexlattice is the next keyword which is also required. This is where the general layout of the core's hexagonal lattice is defined. Figure 1-1 is important as it shows the location in a ring where the indexing starts.

```
core_hexlattice {
    assembly_surf = surface_hex_name
    num_ring      = 3
    fill          = [ assembly_hex_name assembly_hex_name assembly_hex_name ]
    replace_ring( 2 ) = [ assembly_hex_name assembly_hex_name assembly_hex_name
                          assembly_hex_name assembly_hex_name assembly_hex_name ]
    replace{ ring=3 index=18 name=assembly_hex_name }
}
```

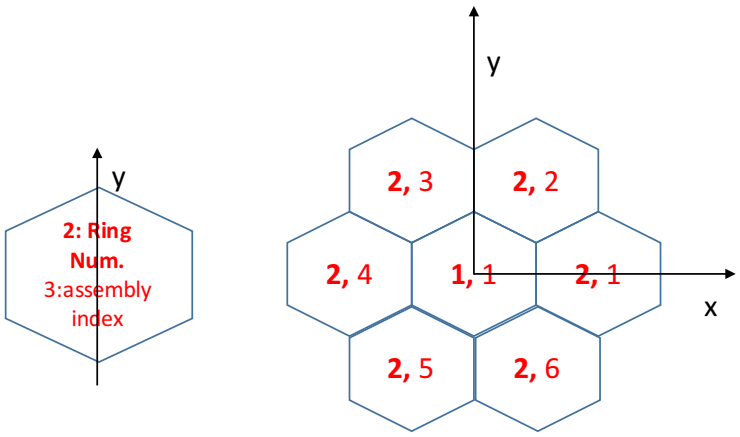

Figure 1-1. Radial layout for the hexagonal lattice.

- **assembly_surf**: Change "surface_hex_name" to the name of the hexagon previously defined under **surfaces** for the assembly boundary.
- **num_ring**: the number of rings in the lattice (center is ring 1).
- **fill**: Assuming for now that in each ring, every assembly in that ring is the same type, **fill** says which type of assembly (defined in Section 1.4.2) is in which ring. The list goes in order starting with ring 1 and moving outward. The details of the different assemblies will be defined later under **assembly_hex**.

Example 3:

```
core_hexlattice {
        assembly_surf = your_hex_surface_name
        num_ring = 5
        fill = [ orange blue green yellow blue ]
}
```

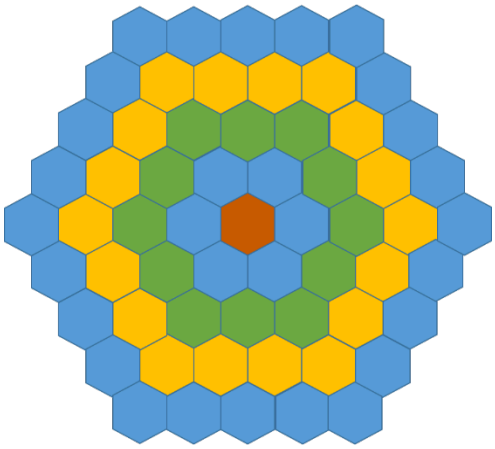

Figure 1-2. Core layout for Example 3.

*In **fill**, the third ring was set to be green assemblies, but this can be overridden with **replace_ring** or **replace** to define any assemblies that are different from the rest of the ring.*

- To define any assemblies that are a different type from whatever was defined for its ring (the "exceptions"), **replace_ring** or **replace** can be used (but not both for the same ring).

Example 4:

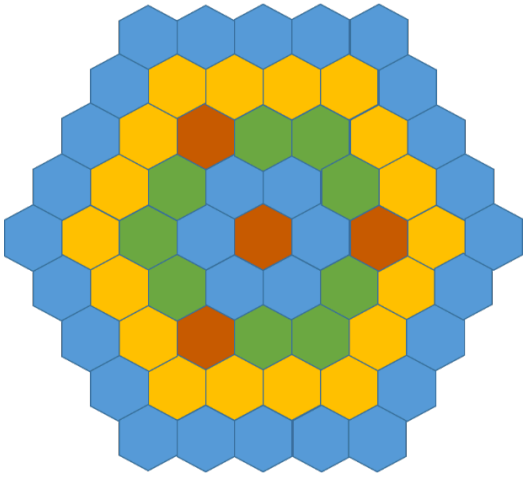

Figure 1-3. Core layout for Example 4.

The layout in the picture above could be described by either:

```
core_hexlattice {
                assembly_surf = your_hex_surface_name
                num_ring = 5
                fill = [ orange blue green yellow blue ]
                replace_ring (3) = [ orange green green green orange green green
green orange green green green ]
        }
```

or

```
core_hexlattice {
                assembly_surf = your_hex_surface_name
                num_ring = 5
                fill = [ orange blue green yellow blue ]
                replace{ ring=3 index=1 name=orange }
                replace{ ring =3 index=5 name=orange }
                replace{ ring =3 index=9 name=orange }
        }
```

1.4.2 assembly_hex

Now that the layout of the lattice is built. The next thing to do is define the details and composition of each assembly. This is done with **assembly_hex**.

```
assembly_hex ( assembly_hex_name ) {
    sub_assembly_hex ( sub_assembly_hex_name ) {
                lower_axial_surf=plane_name1
                upper_axial_surf=plane_name2
                % insert options{}
                % insert materials or assembly_hexlattice
    }
}
```

Change "assembly_hex_name" to one of the assembly types that was used in **fill** ("orange", "blue", "green", or "yellow" for the examples above)

- **sub_assembly_hex**

At least one **sub_assembly_hex** is necessary for every **assembly_hex.** If the assembly is uniform throughout, only one **sub_assembly_hex** is needed. Use more if the assembly has different axial regions.

Change "sub_assembly_hex_name" to a name that describes the region.

*Examples 5-8 demonstrate geometry and materials only, ignore **options** for now.*

Example 5: Uniform homogenous assembly without different axial sections

```
assembly_hex ( uniform_assembly ) {
    sub_assembly_hex ( sub_uniform_assembly ) {
                lower_axial_surf=z0
                upper_axial_surf=z20
                material=material1
                % insert options{}
    }
}
```

*Here z0 and z20 are the top and bottom of the entire region being modeled (including other assemblies), since there are no other **sub_assembly_hex**'s in this **assembly_hex**. z0 and z20 should also be listed as the lower and upper surfaces under **regions_reactor** one level above. This assembly is homogenous and filled with material1.*

Example 6: Different axial sections but each is homogenous

```
assembly_hex ( axial_regions ) {
    sub_assembly_hex ( axial_region_1 ) {
                lower_axial_surf=plane1
                upper_axial_surf=plane2
                material=material1
                % insert options{}
    }
    sub_assembly_hex ( axial_region_2 ) {
                lower_axial_surf=plane2
                upper_axial_surf=plane3
                material=material2
                % insert options{}
    }
    sub_assembly_hex ( axial_region_3 ) {
                lower_axial_surf=plane3
                upper_axial_surf=plane4
                material=material3
                % insert options{}
    }
}
```

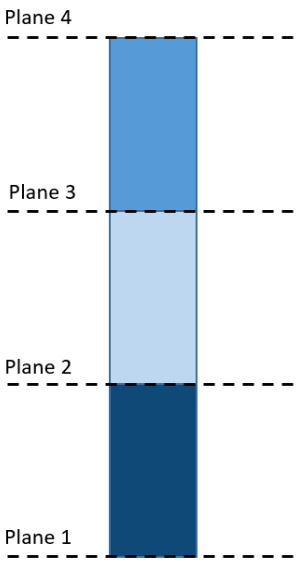

Figure 1-4. Axial regions for Example 6.

Here plane1 and plane4 are the boundaries of the entire model. Each axial region is homogenous with a separate material.

Example 7: One axial section with pins

```
assembly_hex ( assembly_with_pins ) {
    sub_assembly_hex ( sub_assembly_with_pins ) {
                lower_axial_surf=bottom
                upper_axial_surf=top
                % insert options{}
                assembly_hexlattice {
                    pitch         = 0.02            % pitch of the lattice in meters
                    num_ring      = 3
                    outer         = coolant_mat
                    fill          = [ fuel_pin fuel_pin fuel_pin ]

                    pin_region ( fuel_pin ) {
                        sub_pin_region ( fuel ) {
                            material   = fuel_mat
                            outer_surf = fuel_outer         % cylinder name
                        }
                        sub_pin_region ( gap ) {
                            material   = void
                            inner_surf = fuel_outer         % cylinder name
                            outer_surf = cladding_inner     % cylinder name
                        }
                        sub_pin_region ( cladding ) {
                            material   = cladding_mat
                            inner_surf = cladding_inner     % cylinder name
                            outer_surf = cladding_outer     % cylinder name
                        }
                    }
                }

    }
```

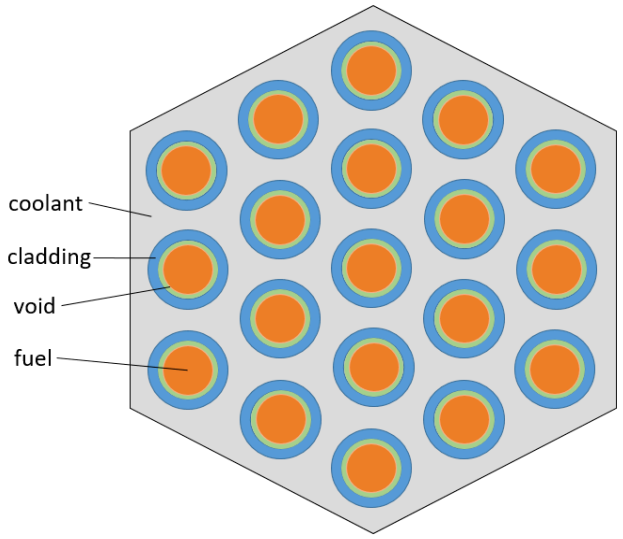

Figure 1-5. Pin lattice for Example 7.

*This example only has one axial region (there is only one **sub_assembly_hex**), so the pins extend the whole length. This geometry can be seen in* Figure 1-5.

When building a lattice of pins:

- Don't include **inner_surf** for the central region.
- **fill** and **replace/replace_ring** are used to define the layout of the pins in the same way as for the assemblies.
- Undefined space between pin regions will be void.
- In some cases, the code may fill undefined regions with the "outer" material for homogenized calculations. **It is recommended that the user not leave any space undefined. One way around this is to make a "void" material that has a very low atom density.**
- For **wire_wrap**, **path** is the height of one turn of the wire.

Example 8: Radial regions

```
assembly_hex ( radial_region_assembly ) {
    sub_assembly_hex ( sub_radial_region_assembly ) {
                 lower_axial_surf=bottom
                 upper_axial_surf=top
                radial_region ( outside ) {
                    material   = material1
                    inner_surf = hex2      % cylinder or hexagon name
                    outer_surf = hex3      % cylinder or hexagon name
                }
                radial_region ( middle ) {
                    material   = material2
                    inner_surf = hex1      % cylinder or hexagon name
                    outer_surf = hex2      % cylinder or hexagon name
                }
                radial_region ( inside ) {
                    material   = material3
                    %inner_surf = surface_name1      % cylinder or hexagon name
                    outer_surf = hex1      % cylinder or hexagon name
                }
                 % insert options{}
    }
}
```

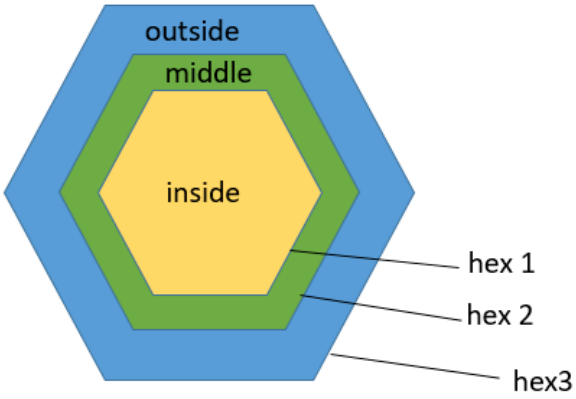

Figure 1-6. Radial regions for Example 8.

radial_region makes rings in the subassembly out of cylinders or hexagons. **radial_region** can be used along with **assembly_hexlattice**. Any undefined space between radial regions will be void.

It is recommended that the user not leave any space undefined. One way around this is to make a "void" material that has a very low atom density.

For explicitely defined sub-assembly (with radial region or hexlattice), the volume fraction of each elementary material calculated in the homogenized material is printed out in the summary file.

1.4.3 options

options is necessary for each **sub_assembly_hex**. It provides options for deterministic codes on the calculations to perform for each subassembly region.

- **mcc3**:

```
options{
   mcc3( id ){
       calculation=mixture
       buckling_search=false
       external_source=id_name
   }
}
```

Replace "id" with either a letter or number between 0 and 9 to identify this calculation. Table 1-1 provides more information about the MCC3 options.

Table 1-1. Keywords under geometry/regions_reactor/assembly_hex/sub_assembly/hex/options/mcc3.

calculation	"mixture" ("cylinder" is also a choice but it is not implemented yet)
buckling_search	generally, "true" for fissile regions if 2-step RZ procedure is not employed, "false" otherwise
external_source	the id of another mcc3 calculation. For a non-fissile region, the user can choose to use the neutron leakage data from another mcc3 calculation as an external source to this mcc3 calculation.

- **mcc3_existing**: If two regions have similar composition, the multi-group cross-sections calculated with MCC3 from one region can be used for the other, instead of doing separate calculations for each subassembly, which saves time. In this case, replace "id_name" with the id of the mcc3 calculation that is to be reused.
- **dif3d**:
  - **num_axial_burnup_zones:** This provides the option to refine the mesh for depletion calculations.

2. calculations

The **calculations** block describes the list of calculations requested by the user and provides main options. It contains the **mcc3** and **dif3d** sub-level, **dif3d** contains also the **rebus** and **persent** sub-levels. It also contains the **num_cpu_max** option which provides the maximum number of CPUs used for parallel computing of different MC^2-3 cell calculations.

2.1 mcc3

Table 2-1 summarizes the options available or required for running MC^2-3. The MC^2-3 code can be run as a stand-alone code to generate multi-group XS, or can be used together with a 2D full-core calculation code (such as TwoDant), which options are listed in Table 2-2.

Table 2-1. Keywords under calculations/mcc3.

xslib	"endf7.0" and "endf7.1" are the only cross section library currently available
egroupname *	if using an energy group structure known by mcc3, choose either "anl33", "anl70", "anl230", "anl1041", or "anl2082"
egroupvals *	if using other energy groups, list the boundaries (not including 0) of the energy groups in eV.
scattering_order	1,3,5… scattering order for multi-group XS
run_mcc3**	"true" runs mcc3, or "false" skips mcc3 if an isotxs file is provided.
lumped_element_text_file	name of file containing composition of the lumped element used in the material definition or in the decay chain
RZ_core_options	This option increases the accuracy of multi-group XS by condensing them based on the correct fine-mesh flux calculated using RZ modeling. See table Table 2-2.

*Use either **egroupname** or **egroupvals** but not both.

**If mcc3 was already performed, the user may not want/need to run it again, in which case the user can use "run_mcc3=false" This would be the case for example if the only changes that were made since the last run related to dif3d and rebus calculations. Then choosing "false" will save the time of running mcc3. When using "false" the user will need to provide an isotxs file for the dif3d, rebus, and/or persent calculations under calculations/dif3d/isotxs. An isotxs file is generated when mcc3 does run, so this file can be re-used.

Table 2-2. Keywords under calculations/mcc3/RZ_core_options.

code	"twodant" ("partisan" is also a choice, but not available yet)
egroupname	"anl33", "anl70", "anl230", "anl1041", or "anl2082" (choose one) This is the fine-group structure that should be used in the RZ calculation. It is recommended to use the ANL1041 group structure. It should contain more energy groups than the one used with **mcc3**.
R_boundaries	boundaries of the different areas in the R direction.
R_nodes_distantance	distance between each node in the R direction, recommended 0.05 m
Z_boundaries	boundaries of the different areas in the Z direction.
Z_nodes_distance	distance between each node in the R direction, recommended 0.08 m
SN_angular_order	angular order for the transport approximation with SN method. Recommended ≥ 12
RZ_geometry	Fill out the areas with mcc3 ID number here to define the RZ core modeled. The number of columns must equal number of R boundaries and number of rows must equal number of Z boundaries. The first row of mcc3 ID's is the bottom of the core and the last row is the top of the core.

How to get an equivalent RZ geometry? calculate the number of assemblies represented by each mcc3_id. Calculate the area for each region: the area of an hexagon is equal to $A = \frac{\sqrt{3}}{2}p^2$ with p the pitch of the assembly. Find the "equivalent radius" by solving for r in $A=\pi r^2$.

2.2 dif3d

The user needs to specify which solver they want to use: DIF3D-FD, DIF3D-Nodal, or VARIANT and provide the associated options. Table 2-3 summarizes the options available or required to specify the DIF3D calculation.

Table 2-3. Keywords under calculations/dif3d.

power	power of the full core in W (doesn't include the symmetry)
geometry_type	hexagonalz_infinite_lattice *, hexagonalz_full_core, hexagonalz_sixth_core**, hexagonalz_third_core**, triangualrz_full_core
isotxs	"previous" tells the code to use the cross sections isotxs file generated from the previously defined mcc3 calculation. If the user already has a valid isotxs file, they can specify it here and disable the MCC3 calculation by using the "run_mcc3 = false" option.
max_axial_mesh_size	maximal axial distance between each calculation node used in the DIF3D model.
dif_fd_options* **	if triangularz geometry is used, the user can specify a triangular subdivision of the assembly mesh.
dif_nod_options* **	the user can provide the option to perform the coarse mesh rebalance.
variant_options* **	The user needs to specify variant option with polynomial and angular approximation and anisotropic scattering. Default values are provided but higher/lower fidelity may be needed depending on the core analyzed.

* hexagonalz_infinite_lattice only works with one subassembly and forces symmetry

** hexagonalz_sixth_core and hexagonalz_third_core will force symmetry even if the whole core that the user built does not have that type of symmetry, without giving any error message. The user should make sure to select the proper geometry type for their problem.

*** can only have one of **dif_fd_options**, **dif_nod_options**, or **variant_options**

2.2.1 rebus

Table 2-4 summarizes the options available or required for REBUS-3, and Table 2-5 lists the requirements for the decay chain.

Table 2-4. Keywords under calculations/dif3d/rebus.

cycle_length	cycle length in equivalent full power days
shutdown_time_between_cycle	shutdown time between cycles in days
num_cycles	number of cycles to be performed
num_subintervals	number of sub-intervals within each cycle
list_materials_depleted	informs which material will need to be depleted
decay_chain	The user can define their own decay chain, in which case he needs to specify everything in Table 2-5, or can use one of the decay chains provided.

Table 2-5. Keywords under calculations/dif3d/rebus/decay_chain.

list_isotopes	all isotopes part of the decay chain (heavy nuclei and fission products if explicitly defined)
list_lumped_elements	names of lumped elements if defined lumped elements are used
list_dummy_elements	name of dummy elements if defined dummy elements are used
text_file	path to decay chain file that provides the decay chain in REBUS-3 format

2.2.2 persent

Persent calculation requires using **variant_options** in dif3d. There are three keywords under persent: **pert_calc**, **sens_calc,** and **uncert_calc**.

pert_calc is used for perturbation theory calculations. It requires to define a perturbed state for the core considered to calculate the difference in reactivity with the reference state. A perturbation calculations requires "pert_id" which can be any available letter. The instructions for **pert_calc** are explained in Table 2-6.

Table 2-6. Keywords under calculations/dif3d/persent/pert_calc.

depletion_step	0 (capability for other time steps not yet implemented)
pert_type	"keff" ("lambda_beta" not implemented yet)
material_perturbation	can define multiple material perturbed, **at least one required**
• **list_isotopes_perturbed**	"all_from_material" or can list any isotopes from the material
• **density_perturbation_factor ***	factor by which the density of the isotopes perturbed will be multiplied
• **new_temp ***	new temperature of the isotopes perturbed (requires xs_updated = true)
xs_updated	"true" or "false" to indicate whether cross-sections will be recalculated for the perturbed state
pert_option	"first_order_perturbation_theory" or "general_perturbation_theory"

* must use **density_perturbation_factor** or **new_temp**, not both.

sens_calc is used for sensitivity calculations on multi-group cross-sections calculated with MC^2-3. Sensitivity calculations can be calculated on the reference core or on a perturbed state. A sensitivity calculation requires a "sens_id", which is any available letter. The instructions for **sens_calc** are explained in Table 2-7.

Table 2-7. Keywords under calculations/dif3d/persent/sens_calc.

depletion_step	0 (capability for other time steps not yet implemented)
sens_type	"keff" ("lambda", "beta" not implemented yet)
sensitivity_regions	list of ids from options/mcc3 on which the sensitivities are calculated
list_isotopes	"all_from_sensitivity_regions" or can list any isotopes in the region
list_reactions	choose any or all reactions within: "fission" "chi" "capture" "elastic" "inelastic" "nu" "n2n" "plelastic"
material_perturbation	**optional**, can define multiple material perturbed to calculate sensitivity on the core in a perturbed state.
• **list_isotopes_perturbed**	"all_from_material" or can list any isotopes from the material
• **density_perturbation_factor ***	factor by which the density of the isotopes perturbed will be multiplied
• **new_temp ***	new temperature of the isotopes perturbed (requires xs_updated = true)
xs_updated	"true" or "false" to indicate whether cross-sections will be recalculated for the perturbed state
sensitivity_factor	factor by which the cross-sections are increased for the sensitivity analysis

* must use **density_perturbation_factor** or **new_temp**, not both.

uncert_calc is used for uncertainty calculations. **These are not implemented yet**. Replace "uncert_id" with any available number. The instructions for **uncert_calc** are explained in Table 2-8.

Table 2-8. Keywords under calculations/dif3d/persent/unert_calc.

sens_ref	takes the id of sensitivity used in a reference-state core
sens_pert	optional, takes the id of sensitivity used in a perturbed-state core
covariance_matrix_text_file	takes the file name of a covariance matrix

Nuclear Engineering Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 208
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC